UPMC Paris Universitas – Master Informatique – STL

Cours Composant
1. Introduction

©2005-2008 Frédéric Peschanski

UPMC Paris Universitas

13 février 2008

Point de vue du cours

Articulation entre :

- Technologies « du moment »
  - Java « avancé », XML (schémas), OSGi, Plugins Eclipse, etc.
- Méthodes « scientifiques » d'ingénierie
  - Spécifications algébriques
  - Conception par contrat
  - Logique de Hoare

Pourquoi des « maths » ?

Constat : méthodes formelles peu visibles

- Conception logicielle : création ou construction ?
- Informaticiens vs. non-informaticiens (ex. UML)
- Méthodes « cadrées » mais empiriques, **qualité** = **test** ?

Mais . . .

- Glissement progressif vers les outils d'aide au développement, basés sur des techniques formelles (ex. Spec#)
- Rappel : le métier d'ingénieur (ex. ingénieur en BTP)
- La sûreté et la sécurité du logiciel,notion plus générale de **qualité logicielle**

En Composant, nous nous intéressons aux méthodes formelles dites « légères » (*lightweight formal methods*), un bon compromis.

Prérequis

- Connaissances techniques :
  - Maîtrise des **concepts objets** (encapsulation, relations, héritage, designs patterns, etc.)
  - Bonne pratique de **Java** - language de support du cours
  - Familiarité avec les technologies **XML** (notamment XSchema)
- Plus fondamentalement :
  - Manipulation des structures discrètes (ensembles, relations, fonctions, etc.)
  - Revoir ses cours de **logique** (pour la logique de Hoare)

Fonctionnement du cours

- Cours
  - Concepts fondamentaux
  - Méthodologie
  - Cas particuliers : JavaBeans, OSGi et Plugins Eclipse
- Travaux dirigés
  - Illustration des concepts
  - Mise en œuvre de la méthodologie
- Travaux sur machine encadrés
  - Acquisition de technologies récentes qui illustre le concept de **composant logiciel**
  - JavaBeans, composants contractualisés (Tamago), OSGi, Plugins Eclipse, Web services

Déroulement des TME

1. Préparation
   - Lecture d'un document de type Tutoriel
   - **Important** : les cours et TD ne préparent pas, en général, aux TME. Notamment concernant les technologies employées.
2. Séance de TME
   - exercice de « rapidité »
   - à la fin des 2 heures : **relevé des TME** (formal jar normalisé)
   - Remarque : préparation nécessaire (point 1)
3. Soumission : version 2
   - Remarque : en général, pas le temps de finir en 2 heures
   - Possibilité de soumettre un TME complété envoyé au plus tard la veille de la prochaine séance

Composant logiciel

Important : notion difficilement réductible à une définition (exercice : définition d'« objet » ?)
Quelques définitions proposées :

- **C. Szyperski** « *Une unité de composition avec des interfaces spécifiées contractuellement et seulement des dépendances vis-à-vis de son contexte. Un composant logiciel doit pouvoir être déployé indépendamment et est l'objet de composition par des tiers* »
- **B. Meyer** « *Un composant logiciel est un élément logiciel (unité de modularité) qui satisfait les trois conditions suivantes :*
  1. *il peut être utilisé par d'autres éléments logiciels, ses clients,*
  2. *il possède un mode d'emploi officiel, qui est suffisant pour que ses clients puissent l'utiliser*
  3. *il n'est pas lié à un client unique* »

Mots-clés

Composition logicielle Conception et implémentation par assemblage (statique ou dynamique) de briques logicielles

Interface Spécification du mode d'emploi associé à un élément logiciel : ce que fait le logiciel indépendamment de comment il le fait

Contrat logiciel Spécification entre le fournisseur du logiciel (qui conçoit et implémente les composants) et ses clients (qui utilise et/ou assemble les composants)

Context-awareness Conception du logiciel en intégrant en amont la spécification explicite des dépendances externes

Architecture logicielle Conception et spécification séparée de l'architecture du logiciel, indépendamment de sa conception et implémentation interne

Déploiement De la conception/implémentation à l'utilisation/exécution

Trustable component

Cours CPS : vers le composant logiciel « de confiance »

- **B. Meyer** « Un *trustable component* est un élement logiciel réutilisable qui spécifie et garantit des propriétés liées à la qualité »

réutilisabilité Critère de qualité. Propriété d'un élément logiciel à être employés dans plusieurs contextes différents, potentiellement pas des clients variés. Important : ce critère ne se décrète pas, on le constate à posteriori

spécification logicielle Document plus ou moins formel qui décrit précisément ce que l'on attend du logiciel, tant en termes de conception que d'implémentation

garantie de qualité cf. cours 2

Composants CPS

Dans ce cours, on retient principalement de ce vaste programme :

Check-list Composant logiciel

- Les **interfaces** spécifient clairement ce que fait le composant
  - Interface interne : fonctionnalités du composant
  - Interface externe : spécification du contexte d'utilisation (dépendances explicites) : le composant a vocation à être **composé** avec d'autres composants
- **Garanties** sur les implémentations
  - Relie la spécification et ses possibles implémentations
  - Approche « légère » : conception par contrat
  - Approche « lourde » : logique de Hoare
- **Unité de déploiement** : packaging, versioning, chargement, cycle de vie, etc.

Remarque : objets vs. composants

Les objets sont des composants « minimaux » :

- Encapsulation, cycle de vie minimal (création, utilisation, destruction)
- Interface interne minimale : classe et/ou interface

Mais il manque (entre autre) :

- des spécifications plus « sémantiques »
- l'explicitation du contexte d'utilisation (interface externe)
- les garanties sur la qualité (exception : conception par contrat)
- la problématique du déploiement

Plan des cours

1. Designs patterns pour les composants
2. Qualité logicielle, spécifications algébriques
3. Conception par contrat I
4. Conception par contrat II
5. Interlude 1 : déploiement avec OSGi
6. Logique de Hoare I
7. Logique de Hoare II
8. Interlude 2 : plugins Eclipse
9. Concurrence et coordination
10. Interlude 3 : services Web et processus métiers

Le « noyau dur » du cours

Noyau dur

Méthodologie pour la conception et l'implémentation de **trusted components**

1. **Analyse** : spécifications algébriques (cours 2)
2. **Conception** : par contrat à partir des spécifications (cours 3 et 4)
3. **Implémentation** : garantie vis-à-vis des contrats
   - Conception par contrat : vérification « légère » en ligne (self-testing)
   - Logique de Hoare : vérification « lourde » à priori (cours 6 et 7)

Plan du cours 1

Designs patterns pour les composants

- Patterns des Javabeans
  - Pattern : « Observable properties »
  - Pattern : « Event-based communication connector »
- Pattern : « Require/Provide connector »

Propriétés javabeans

Motivation : Observabilité

Un composant doit être capable d'expliciter ce que l'on peut observer sur son état interne. Un composant doit être également configurable.

Sinon on ne peut rien spécifier à l'avance sur l'état, et rien n'est vérifiable ensuite.

Pattern : « Observable properties » (Javabeans)

Propriété observable définie par :

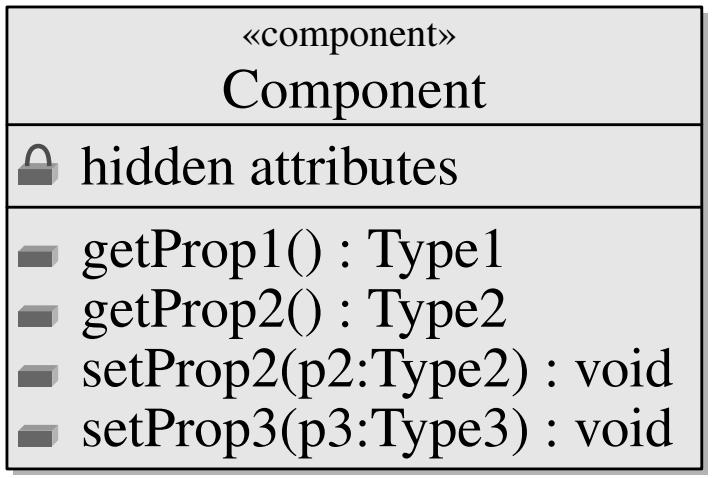
- un **nom**
- un **type**
- un **mode d'accès** : lecture seule, lecture/écriture, écriture seule (configuration)

Structure du pattern

Exemple JavaBean

Propriété de nom `prop` de type `Type` en lecture/écriture

```
// Access : read
public Type getProp() ;
// Access : write
public void setProp(Type value) ;
```



Javabean : exemple

```
public class Interrupteur {
   public enum { ON, OFF } State;
   private boolean internalState;
   public Interrupteur() { internalState = false; }

   /* Propriétés */
   public State getState() {
      if(internalState==true)
         return ON;
      else return OFF;
   }

   /* Fonctionalités */
   public boolean isOn() { return state==true; }

   public void switch() { state = !state; }
}
```

Autres exemples : composants Swing

Bilan pattern *Observable properties*

Intérêts

- Observabilité : concept fondamental (plus qu'il n'y paraît)
- Mise en œuvre simple

Limites

- En lui-même le pattern ne fait pas grand chose, il faut des outils pour l'exploiter
- Difficultés « cachées » : types observables

Pattern : « Event-based communication connector »

Motivation

Un composant doit expliciter ses dépendances externes
$\Rightarrow$ notion de **connecteur logiciel**

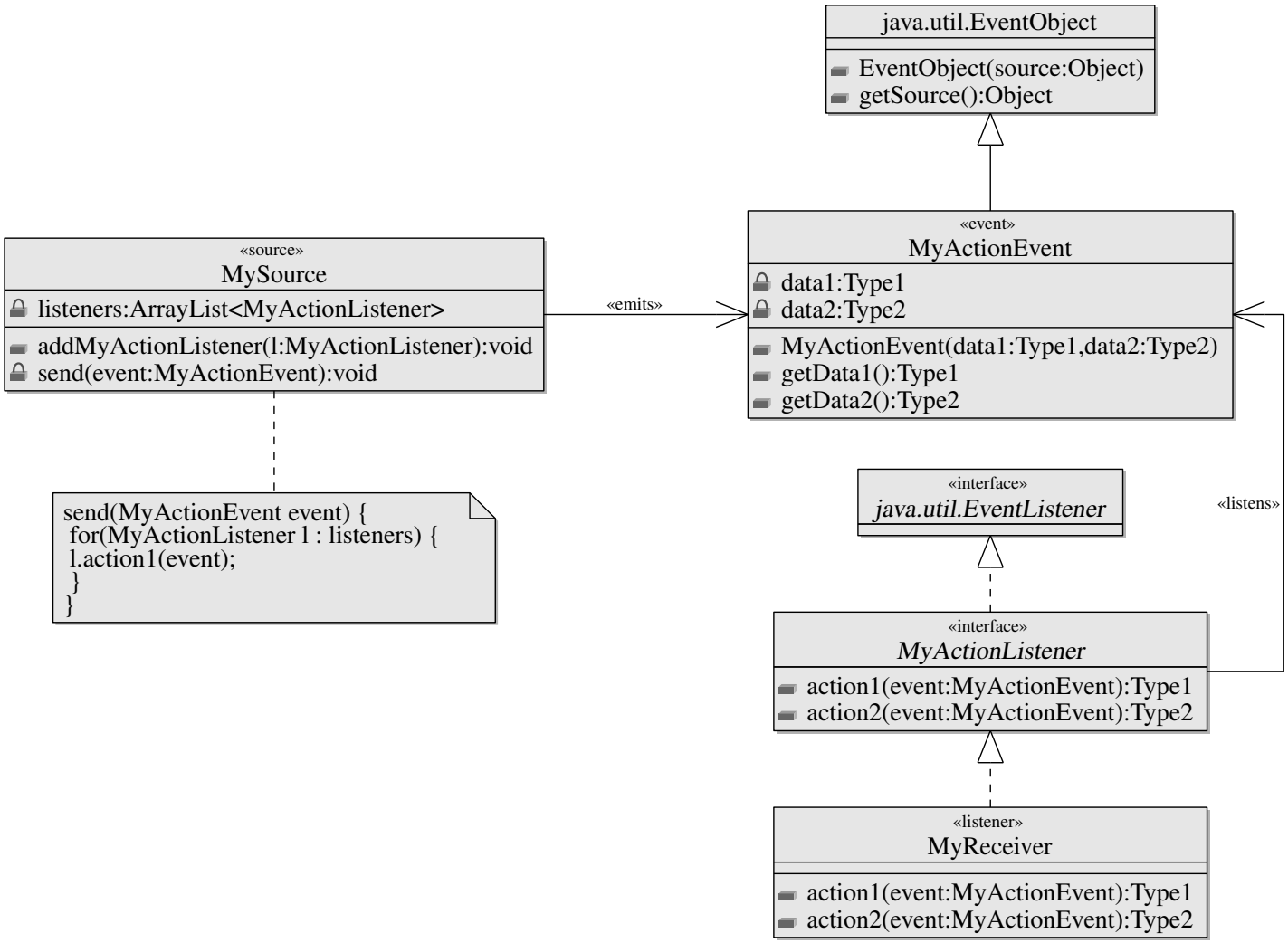
Javabeans : mode de communication évenementiel

- Composant : **source** et/ou **écouteur** d'événements
- **Événement** : objet passif et immuable

Constituants

- Classes (types) d'événements : `MyEvent`
- Interfaces d'écoute (*listener* : `MyEventListener`)
- méthodes de réaction : dépend de l'événement

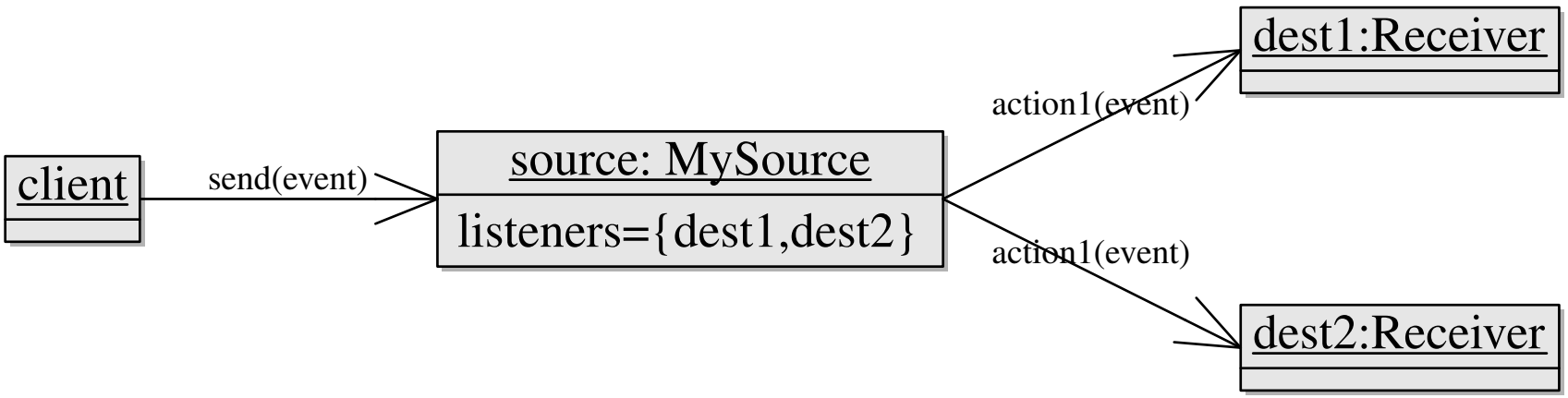
Structure du pattern



Evénement : exemple d'occurence

Préalabe : les deux écouteurs sont enregistrés auprès de la source

```
source.addMyActionListener(dest1) ;
source.addMyActionListener(dest2) ;
```



Bilan pattern *Event-based communication*

Intérêts

- Découple structurellement les émetteurs et récepteurs d'événements
- Permet la communication « 1 (émetteur) vers N (récepteurs) »
- Explicite la dépendance du récepteur (interface d'écoute)
- Evénements multiples et sélection par typage
- Branchements dynamiques (si nécessaires)

Limites

- Pas de découplage du contrôle : la source doit contrôler la communication
- Dépendances explicitées seulement dans un sens : le récepteur est identifié comme tel, mais pas la source
- Pattern un peu « lourd »

Pattern : « Require/Provide connector »

Motivation

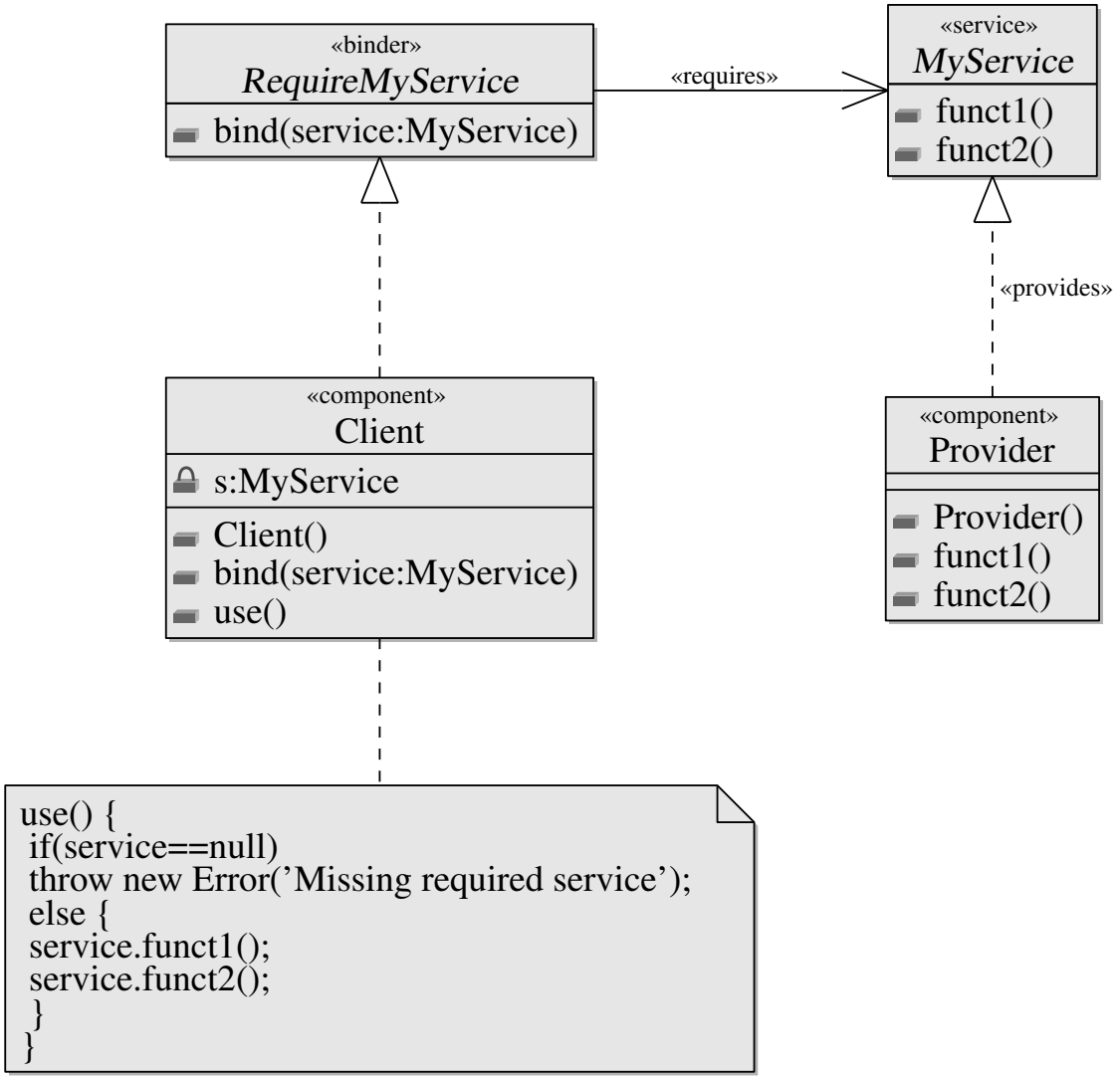
Un composant doit expliciter ses dépendances externes (bis)
Description séparées des fonctionalités implémentées par les composants ⇒ notion d'**interface** ou de **service**[a]

[a] Ne pas confondre « interface de composant » et « interface java ».

Principes

- Un **service** regroupe des fonctionnalités
- Client du service : interface de liaison « **require** »
- Fournisseur de service : implémente le service

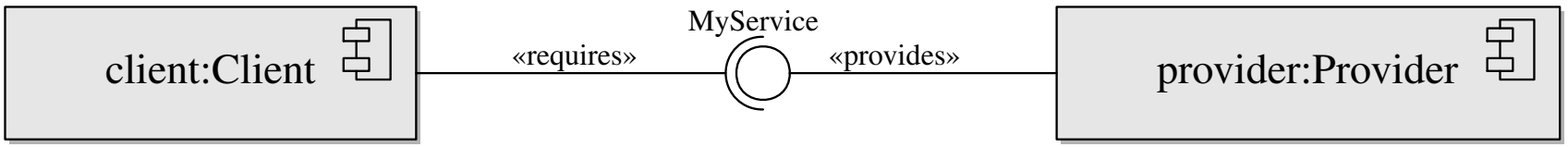
Structure du pattern



Modèle de composition

Composition d'une architecture :

```
Client client = new Client();
Provider provider = new Provider();
client.bind(provider);
client.use();
```



Bilan pattern *Require/Provide*

Intérêts

- Découple structurellement les fournisseurs et clients de service
- Dépendances explicitées dans les deux sens (client et fournisseur)
- Métaphore du service et client/fournisseur (cf. conception par contrat)

Limites

- Pas de découplage du contrôle : le client gère la liaison
- Pattern un peu « lourd » (beaucoup d'interfaces en Java) (mode actuelle : code java simple mais XML pour décrire les liaisons. cf. plugins eclipse)

Questions ?